Measuring T2 against SBST 2013 Benchmark Suite

I.S.W.B. Prasetya
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
Email: s.w.b.prasetya@uu.nl

***Abstract*—T2 is a light-weight, on-the-fly random-based automated testing tool to test Java classes. This paper presents its recent benchmarking result against SBST 2013 Contest's test suite.**

***Index Terms*—testing, automated testing, OO testing, testing tool benchmark**

I. Introduction

In 2013, the International Workshop on Search-Based Software Testing (SBST) organized a tools competition for unit testing of Java programs. The unit-testing is set to be at the *class level* (the unit to be tested is a class as a whole, as opposed to method-level unit testing). Participating tools are applied on a benchmark suite, which we will refer to as *SBST2013-Suite*, consisting of a set of 77 java classes taken from open-source projects [1]. The tools are expected to generate test-suites for each class, and will be measured for their generated coverage as well as their execution overhead. A single-value function, which we will refer to as the *SBST2013 score-function*, is defined to express tools' overall performance.

T2 is a light weight on-the-fly random-based testing tool for Java [2]. It was originally developed as a research tool to allow students to experiment with implementing and improving existing testing techniques. It has become mature enough to be released for public use. T2 however lacks an objective measurement of its effectiveness. Previous measurement [3], [4] were not so extensive and were setup against examples biased towards T2's algorithm. The SBST 2013 Contest is therefore a welcomed opportunity. This paper presents T2's result. The specific way that SBST2013 score-function is defined is unfortunately incompatible with T2's on-the-fly modus, and consequently gives a wrong indication for its performance. We will therefore analyze the benchmark's raw data instead.

Prerequisites	
Static or dynamic	dynamic testing on ADT-like Java class
Software Type	Java
Lifecycle phase	development-time unit testing
Environment	none required, IDE preferred
Knowledge required	intermediate-level Java programming
Experience required	intermediate-level Java programming
Input and Output of the tool	
Input	none
Output	Execution report (text) and saved test-sequences file
Operation	
Interaction	just run and inspect the result
User guidance	none, but some work may be needed to match and cofigure it; see also Section II-B
Source of information	code.google.com/p/t2framework
Maturity	released research tool
Technology behind the tool	reflection
Obtaining the tool and information	
License	GPL version-3
Cost	free
Support	none
Does there exist empirical evidence about:	
Effectiveness	to limited extent: [3], [4]

TABLE I
Description of the tool that is being evaluated

II. How T2 Works

T2 is a tool to do automated unit testing of a Java class [2]. T2's model of a 'class' is that of an Abstract Data Type (ADT). Such a class C specifies: (1) instances of C, each carries its own state, (2) a set F_C of fields accessible by C's clients —they constitute each instance's state, and (3) a set M_C of methods operating on each instance's state. These methods and field-updates typically interact through the instance's state. T2 focuses on exposing errors due to such interactions (as opposed to testing one method at a time). When given such a target class, T2 exploits Java reflection to infer F_C and M_C. It then creates a large number of set of test-sequences, intended to expose faulty interactions. Each sequence starts with a call to C's constructor to create an instance of it, and each step in the sequence is either an update to a field in F_C, and a call to a method in M_C. The selection as well as the parameters of these updates and calls are random, although T2 also has several options to control how test-sequences is generated.

Most Java classes that people write fits naturally into T2's ADT assumption. But there are 'classes' that do not fit; e.g. a class that is merely used to group static methods. Such a class does not have any instance, nor does it have a natural concept of inter-methods interactions. Class-level unit testing on such a class is thus insensical.

A. On-the-fly development-time testing

T2 generates and executes test-sequences in the same run (as opposed to doing these in two separate stages). In other words, the sequences are generated on-the-fly. Combined with its light weight random-based algorithm, such on-the-fly testing is pretty fast. Each T2 run typically only takes few seconds. This

makes T2 to suit well for quick development-time testing. As a programmer is developing a class C, after each modification or refactoring he can use T2 to 'stress' the target class by generating a large amount of random test-sequences to trigger erroneous inter-method interactions.

In such a working mode, short response time is desirable; waiting too long (e.g. minutes) would distract the programmer too much, and annoy him.

T2 does not have the ability to infer test-oracles; so, only thrown (and uncaught) exceptions can be recognized as errors. Due to the randomness of T2, supplying it with concrete values oracles (e.g. $return{==}1.01$) will not work. It should be used ala property-based testing, comparable to the common use of QuickCheck [5], where programmers need to supply correctness properties, encoded as Java assertions. They do not need to be traditional complete pre/post-conditions; few simple assertions cleverly inserted within the body of a method can also be strongly fault revealing. Assertions can be marked as assumptive (pre-condition like), conclusive (post-condition like), or as a class-invariant. T2 will interpret them accordingly: violating a conclusive assertion counts as an error, but violating an assumptive one does not. The latter case means that the current test-sequence is inappropriate and should be aborted.

Given a class C to develop and test, and after the programmer has put some initial assertions in C, the typical work cycle of using T2 proceeds as follows. After each refactoring or a round of modifications on C:

1) The programmer runs T2 to stress the target class to trigger erroneous inter-method interactions.
2) When an exception is thrown, the programmer investigates if it is a real error. He should then fix the error. Else, the exception is caused by an inappropriate call to a method: it violates the method's implicit pre-condition. The programmer needs to make this assumption sufficiently explicit to block T2 from making further inappropriate calls. This is written in a plain Java assertion, marked as assumptive.
   We cycle back to step-1.
3) If no exception is thrown, the programmer either improve or add the assertions, or he is satisfied; then he is done for this round.

Since a benchmarking is typically fully automated, applying the above work cycle is not possible. So, with the SBST benchmarking we will only measure T2's ability in the step-1 above. Morever, we assume SBST2013-Suite to contain no assertions. It also turned out that the number of exceptions thrown were not logged. So we can only approximate the effectiveness in terms of traditional coverage measures.

B. T2 Algorithm

Given a target class C, T2 uses reflection to collect (C_C, M_C, F_C) which are sets of C's constructors, methods, and non-static fields, which are considered to be within the testing 'scope'. This *scope* is configurable, e.g. to include only public members. By default, M_C also includes C's static methods, but only those that take a C as parameter (and thus can potentially affect the state of an instance of C).

A single T2 run generates up to some N test-sequences of length K; these numbers are configurable. Each sequence σ is generated (and executed) as follows:

1) A constructor from C_C is randomly chosen as the first step of σ, and is *executed* to create an instance o of C.
2) After sucessfully extending σ with k steps, a field from F_C is chosen, or a method from M_C is chosen as its $k{+}1$-th step. If a field is chosen, the corresponding field in o is randomly *updated*. If a method is chosen, parameters are randomly created, and the method is *called*, with o as its target. A static method is only selected if it can accept o as a parameter.
   a) If the execution of this $k{+}1$-th step causes an assumptive assertion to be violated, or an `IllegalArgumentexception` to be thrown, then the step is inappropriate. In the T2's *default mode* the entire σ is then aborted. In the *search mode* only the step is aborted, and a new one is tried, up to some x number of attemps.
   b) If the step violates a conclusive assertion, or it causes an uncaught exception to be thrown, it is considered as exposing an error. The sequence σ is then reported, or saved (or both).
3) Depending on T2's configuration it can be made to stop after the first violation, or to continue generating the next sequences, and to stop after collecting some v number of exposed errors.

To generate parameters of the steps in a test-sequence, T2 works as follows. Primitive values are randomly chosen from a so-called *base-domain*. Objects are either generated fresh, by recursively calling constructors, or by reusing previously created objects (to trigger inter-steps data coupling). If multiple options are possible, one is chosen randomly. A default simplistic base-domain is provided, which can be refined through sub-classing to improve T2's effectiveness towards a particular target class C. Also, a custom *interface-map* can be defined to help T2 in selecting the right subclass when it has to create an object.

C. T2 configuration for the benchmarking

On every target class C, we configure T2 to consider C's non-private members to be within the testing scope. Private members are not meant to be called by C's environment, so calling them from a test-sequence would be inappropriate. No custom base-domain nor interface-map is used. We set T2 to do seven runs on C, of different configurations as shown in Figure 1; each run generates the corresponding test-suite.

The last run is set on T2's combinatoric mode, which can be used to generate k-combinations of M_C as test-sequences; see also [6]. Unfortunately, we later discovered that it was set with wrong parameters that rendered it ineffective.

SBST's benchmarking procedure insists on separating the test generation and test execution phases. Coverage is measured during the second phase. The two phases are actually

test-suite	mode	N	K
set-std-3	standard	1000	3
set-search-3	search	1000	3
set-std-10	standard	1000	10
set-search-10	search	1000	10
set-std-20	standard	500	20
set-search-20	search	500	20
set-combin-8	combinatoric	-	-

Fig. 1. *Seven test-suites generated by T2 for each target class C. N is the maximum number of test-sequences (inappropriate test-cases are dropped), and K is the maximum length of each test-sequence (it can be shorter if it discovers an error before the K-th step).*

inseparable in T2. So, to get coverage results we decided to make a 'dummy' execution phase. Each run above will save the generated set of test-sequences. Only unaborted sequences will be saved, and they are saved irrespective of whether or not they trigger a non-assumptive exception. In favor for speed, T2 saves the set in a single binary file. During the dummy execution phase, we simply let JUnit to replay the saved test-sequences.

III. Result

SBST2013-Suite consists of 77 classes, taken from various open source projects. Against this suite T2 scores 50.4938, calculated by this function [1]:

$$score = \Sigma(cov_l + 2*cov_b + 4*cov_m) - 5*(t_{prep} + \Sigma(t_{gen} + t_{exec}))$$

where cov_l, cov_b, cov_m are line, branch, and mutation coverage; and $t_{prep}, t_{gen}, t_{exec}$ are respectively the time it takes for the tool to prepare (e.g. to do a preliminary analysis), to generate the test-cases, and to execute the test-cases.

This rating function turned out to be unreasonable for T2. Firstly, it heavily favors the coverage component; in particular the mutation component. Despite the given weight of 5, the contribution of the time-component is actually almost negligible because it is calculated in hours. Whereas for an on-the-fly testing tool like T2, response time is actually an important indicator. Furthermore, the measured execution time acutally belongs to the *replay* of the saved test-suites (see also Subsection II-C). These suites were already executed when they were generated. In other words, T2's t_{gen} already includes t_{exec}.

Secondly, the mutation score cov_m is measured with respect to a class C is calculated over the test-cases that do not throw an exeception on the original/unmutated C. The premise is thus that C is *already correct*, and that the task of the tool is to generate regression test-cases. This is quite the opposite of T2's normal use (Subsection II-A), where C would *not* be assumed to be correct; we use T2 to stress C to try to trigger exceptions, rather than to avoid them! A reverse-mode of this usage is unfortunately unsupported. Furthermore, T2 normally only saves violating test-sequences for the purpose of debugging the found error. As pointed out in Subsection II-A, to get coverage reading we made T2 to save *all* generated test-sequences. In favor for speed, T2 saves them in a binary file. In particular, it does not generate a JUnit test method per test-sequence. Unfortunately, from the perspective of SBST's benchmarking procedure, this makes a whole test-suite to be treated as a single test-case. So, if one test-sequence in the suite throws an exception, the whole suite is disqualified. Effectively this means that T2's mutation score will always be 0.

	t_{resp}	cov_b
Manual	-	0.82
EvoSuite	186.3	0.58
Randoop	101.2	0.40
T2	7.8	0.23

Fig. 2. *Response time and branch coverage over the whole SBST2013-Suite*

Fortunately, SBST provides some raw data of the experiment, and from there we can calculate some indicators. We can still use the data about branch coverage. Line coverage data are also available, but it seems to be unreliable: in some cases we see 0 line coverage, whereas the corresponding branch coverage is non-zero. Additionally, for an on-the-fly tool like T2, its response time is important. Figure 2 shows how it performs, compared to manually crafted test-suite, EvoSuite and Randoop.

The response time t_{resp} is defined to be the same as t_{gen} data we obtained from SBST. For T2 this is justified. For the other tools we should actually add their execution time, but these are negligible compared to their t_{gen}. For manual test-suite, no data about the time needed to craft them are available.

Not all classes in the SBST2013 should actually be considered as appropriate targets for T2. Clases that are not ADT-like (see the beginning of Section II) will simply be ignored. Classes with only private constructors will also be ignored, since our configuration exclude private members (Subsection II-C). ADT-like abstract classes and classes implementing singleton patterns should arguably fall within T2's automation range, but no support is currently provided to handle these.

Figure 3 shows T2's result on a set of 40 classes from the SBST2013-Suite where T2's line or branch coverage is non-zero, implying that T2 appears to be able to handle them. This set will be called *40-T2-Suite*. Of these 40 classes, eight are actually inappropriate targets for T2 (as meant above): `org.apache` classes are contains only static methods, `MillisDurationField` is an abstract class, and `DurationField` only has a private constructor. `ISOPeriodFormat` and `ISODateTimeFormat` have one trivial constructor, but the rest consists of static non ADT methods. `EnvironmentFactory` is an interesting case. It has one trivial constructor. The rest are static methods that are also coupled through static fields. It is thus analogous to a singleton ADT. Arguably T2 should be able to deal with that; but it can't. If we consider these classes as 'inappropriate' for T2 as it is now, we are the left with 32 valid targets, which we will call *32-T2-Suite*. Finally, those classes marked with /C in Figure 3 are classes that all tools (T2, EvoSuite, and Randoop) are commonly have non-zero line or branch coverage; there are 24 of them. This will be called *24-Common-Suite*.

Figure 4 shows T2 performance with respect to the subsets

	t_{resp}	cov_b
net.sourceforge.barbecue.BlankModule /C	4.80	1.0000
net.sourceforge.barbecue.SeparatorModule /C	4.72	1.0000
net.sourceforge.barbecue.env.DefaultEnvironment /C	6.45	1.0000
net.sourceforge.barbecue.env.EnvironmentFactory /C	6.76	0.0000
net.sourceforge.barbecue.env.HeadlessEnvironment /C	6.02	1.0000
net.sourceforge.barbecue.linear.ean.UCCEAN128Barcode /C	11.84	0.2237
net.sourceforge.barbecue.output.GraphicsOutput	56.26	0.6000
org.apache.commons.lang3.ArrayUtils /C	6.90	0.0000
org.apache.commons.lang3.BooleanUtils	6.69	0.0000
org.apache.commons.lang3.math.NumberUtils /C	6.77	0.0000
org.joda.time.Chronology /C	1.31	1.0000
org.joda.time.DateTimeComparator /C	16.40	0.1993
org.joda.time.DateTimeFieldType /C	1.36	1.0000
org.joda.time.DateTimeUtils /C	6.72	0.0263
org.joda.time.DateTimeZone	1.72	0.1087
org.joda.time.DurationField	1.26	0.0000
org.joda.time.DurationFieldType /C	1.44	1.0000
org.joda.time.chrono.GregorianChronology	1.75	0.1786
org.joda.time.chrono.ISOChronology /C	1.74	0.4375
org.joda.time.convert.CalendarConverter	24.00	0.4000
org.joda.time.convert.ConverterManager /C	11.87	0.4355
org.joda.time.convert.ConverterSet	32.41	0.5250
org.joda.time.convert.DateConverter /C	19.86	1.0000
org.joda.time.convert.LongConverter /C	19.45	1.0000
org.joda.time.convert.NullConverter /C	8.14	1.0000
org.joda.time.convert.ReadableInstantConverter	23.19	0.2500
org.joda.time.convert.ReadablePartialConverter	25.46	1.0000
org.joda.time.convert.ReadablePeriodConverter /C	8.26	1.0000
org.joda.time.convert.StringConverter	8.40	0.0000
org.joda.time.field.BaseDateTimeField	6.88	0.0139
org.joda.time.field.FieldUtils	1.24	0.1006
org.joda.time.field.MillisDurationField	1.22	0.0000
org.joda.time.field.OffsetDateTimeField	41.28	0.5000
org.joda.time.format.DateTimeFormat /C	6.81	0.0000
org.joda.time.format.DateTimeFormatter	21.51	0.2206
org.joda.time.format.DateTimeFormatterBuilder	14.82	0.6019
org.joda.time.format.ISODateTimeFormat /C	6.76	0.0000
org.joda.time.format.ISOPeriodFormat /C	6.85	0.0000
org.joda.time.format.PeriodFormat /C	6.70	0.0000
org.joda.time.format.PeriodFormatterBuilder /C	10.92	0.7342

Fig. 3. *T2's results, restricted on the classes whose line coverage or branch coverage is non-zero.*

	t_{resp}	cov_b
Full SBST-Suite	7.8	0.23
40-T2-Suite	11.4	0.44
32-T2-Suite	12.3	0.55
24-Common-Suite	7.9	0.54

	t_{resp}	cov_b
Manual	-	0.88
EvoSuite	190.5	0.75
Randoop	101.1	0.71
T2	7.9	0.54

Fig. 4. *The left table shows T2's response time and branch coverage over various subsets of SBST2013-Suite. The right table shows tools' response time and branch coverage over the whole 24-Common-Suite.*

defined above, and its performance compared to EvoSuite and Randoop with respect to the 24-Common-Suite. In the next section we will give our conclusions from these results.

IV. Conclusion

Due to its simplistic random-based algorithm, we expected T2 to score lower coverage. Indeed it does. When compared only against ADT-like classes (32-T2-Suite), or the classes that all tools can commonly handle (24-Common-Suite) it does not perform that bad. If we pretend these suites to be the real classes that we have to test, the results suggest that with almost no programming effort at all, the programmer can get in average 55% branch coverage. Our guess is that providing a custom base-domain and adapter will significantly improve the coverage. This is of course manual work that the programmer has to invest, but it need only be done once per class.

The response time is measured to be between 8 - 12 seconds. For on-the-fly testing, where we ideally want to invoke T2 after every modification/refactoring, the response time should ideally be no more than few seconds. However, for this benchmarking we have configured T2 to try to generate all of the specified number of test-sequences. This takes 8 seconds. It can be pointed out that we typically let T2 to stop after the first error it reports —in this situation, the response time would be much shorter. But still, if no error is found, the full suite would have been generated. Future work could be to parallelize T2's engine. Since test-sequences do not in principle interact, generating them on multiple cores should in theory work well.

The results also point out that pure random-based testing should not be expected to replace manual testing. A tool like T2 can best be exploited as a cheap and fast way to smoke out errors, but after that manual testing is still needed to check if more subtle interactions are also correct.

The large difference in T2's coverage over the full SBST2013-Suite (23%) and over e.g. 32-T2-Suite (55%) indicates that perhaps it is useful to extend T2 to also support non-ADT classes. This issue is more an engineering problem. Similarly, providing a reverse-mode to look for non-violating test-sequences can also be done. From this perspective, the Contest has been useful to point out practical features that are missing from the tool.

For future benchmarking we recommend to group the classes in categories, e.g. classes that are ADT-like, abstract classes, classes implementing the singleton pattern, classes that consist mainly of static methods, and so on. This allow tool developers to beter identify the strength and weakness of their tools. Furthermore, the weight of the components need to be rebalanced —finding a right balance is indeed an interesting question.

Acknowledgment

The author would like to thank the SBST 2013 Contest team for providing this experiment –it has been a welcomed opportunity to test out our tool–, and Sebastian Bauersfeld for his assistance and patience.

References

[1] S. Bauersfeld, T. Vos, K. Lakhotiay, S. Poulding, and N. Condori, "Unit testing tool competition," *International Workshop on Search-Based Software Testing (SBST)*, 2013.

[2] I. Prasetya, T. Vos, and A. Baars, "Trace-based reflexive testing of oo programs with t2," *1st Int. Conf. on Software Testing, Verification, and Validation (ICST)*, pp. 151–160, 2008.

[3] ——, "Trace-based reflexive testing of oo programs," Dept. Inf. & Comp. Sciences, Utrecht Univ., Tech. Rep. UU-CS-2007-037, 2007. [Online]. Available: www.cs.uu.nl/research/techreps/UU-CS-2007-037.html

[4] M. Gerritsen, "Extending T2 with prime path coverage exploration," Master's thesis, Dept. Inf. and Comp. Sciences, Utrecht Univ., 2008, available at: http://www.cs.uu.nl/wiki/WP/T2Framework.

[5] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *ACM Sigplan Int. Conf. on Functional Programming (ICFP-00)*, 2000. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[6] I. Prasetya, *T2 : Automated Testing Tool for Java, User Manual (for T2 version 2.2.1)*, 2013. [Online]. Available: http://code.google.com/p/t2framework/